Ex : 9b

Title: Design, Develop and Implement a Program in C to read a sparse matrix to search a particular value and to display the transpose of the matrix in its triplet format.

Problem Description: To manage sparse matrices using a triplet representation and finding its transpose.

Method: The Program must allow the user to read a sparse matrix, display it in triplet format, search for specific values, and transpose the matrix.

Theory Reference: Module 1

Explanation:

Structure Definition

```
struct sparse
{
    int row; // Row index of the non-zero element
    int col; // Column index of the non-zero element
    int val; // Value of the non-zero element
} s[10];
```

This structure represents a non-zero element of a sparse matrix, storing its row index, column index, and value. An array s of this structure is used to store the matrix.

Algorithm:

Step 1: readsparsematrix Function

Reads a sparse matrix from user input and stores its non-zero elements in triplet form.

Step 2: triplet Function

Displays the triplet representation of the sparse matrix.

Step 3: search Function

Searches for a specified key (value) in the sparse matrix.

1. Prompt the user to enter a value `key` to search for.
2. Initialize a flag `found` to 0.
3. Loop from `1` to `s[0].val`:
    - If `s[i].val` matches the `key`:
        - Print the row and column of the found element.
        - Set `found` to 1 and break the loop.
4. If `found` is still 0 after the loop, print "element not found."

Step 4: transpose Function

1. Create a temporary structure array `trans` to store the transposed matrix.
2. Set the dimensions of the transposed matrix:
    - `trans[0].row` = `s[0].col` (new number of rows)
    - `trans[0].col` = `s[0].row` (new number of columns)
    - `trans[0].val` = `s[0].val` (same number of non-zero elements)
3. Initialize a variable `k` to 1 to index into the transposed array.
4. Loop over each column iii of the original matrix:
    - For each non-zero element in `s`:
        - If the column index matches iii:
            - Assign the transposed values:
                - `trans[k].row` = `s[j].col`
                - `trans[k].col` = `s[j].row`
                - `trans[k].val` = `s[j].val`
            - Increment `k`.
5. Copy the transposed matrix back to `s` (overwriting the original).
6. Call `triplet()` to display the transposed matrix.

Step 5: main Function

1. Call `readsparsematrix()` to read the sparse matrix from user input.
2. Call `triplet()` to display the original sparse matrix.
3. Call `search()` to allow the user to search for a specific value in the matrix.
4. Call `transpose()` to compute and display the transposed matrix.

Ex : 10

Title: Design and develop a Program in C that uses Hash function H: K ->L as H(K)=K mod m and implement hashing technique to map a given key K to the address space L. Resolvethe collision (if any).

Problem Description: Given a set K of Keys (4-digit) which uniquely determines the records in file F. Assume that file F ismaintained in memory by a Hash Table (HT) of m memory locations with L as the set of memory addresses (2-digit) of locations in HT. Let the keys in K and addresses in L are Integers.

Method: Ensure that the program generates the 4-digit key randomly and generates the L using hash function, demonstratesif there is a collision and its resolution by using linear probing.

Theory Reference: Module 5

Explanation:

1. **Linear Probing**: Collisions are resolved by checking the next available index (index1 = (index1 + 1) % m;), which continues until an empty slot (-1) is found.
2. **Memory Allocation**: The hash table (a) is dynamically allocated using malloc.
3. **Handling Table Full Condition**: If the hash table is full, it stops inserting further keys and reports the issue.

Algorithm:

Step 1: Initialize the Hash Table

1. **Create an array** `a` of size `m` (in this case, `m = 20`), and initialize all elements to `-1` to represent empty slots.
2. Set `count = 0` to keep track of the number of elements inserted into the hash table.

Step 2: Input the Number of Keys and Validate

1. Ask the user for the number of keys to insert into the hash table.
2. **Validate the input**:

   - If the number of keys `n` is greater than `m`, print an error message and terminate the program (since the hash table cannot accommodate more keys than its size).

Step 3: Input the Keys

1. Ask the user to input `n` keys (each key is a 4-digit integer).
2. Store the keys in an array `key[20]`.

Step 4: Insert Keys into the Hash Table Using Linear Probing

For each key in the array `key[]`:

1. **Compute the index** where the key should be inserted:
   - Use the hash function: `index1 = key % m` (this gives an index in the range `0` to `m-1`).
2. **Linear Probing** to resolve collisions:
   - Check if the slot at index `index1` is occupied (`a[index1] != -1`):
     - If occupied, move to the next slot: `index1 = (index1 + 1) % m`.
     - Repeat this process until an empty slot is found (i.e., `a[index1] == -1`).
3. **Insert the key**:
   - Once an empty slot is found, insert the key at that index: `a[index1] = key`.
   - Increment the `count` to indicate that a key has been successfully inserted.
4. **Check if the hash table is full**:
   - If `count == m`, print a message indicating the table is full and stop inserting further keys.

Step 5: Display the Hash Table

1. Traverse the array `a[]` and print the keys stored in the hash table. For each index:
   - If the slot is empty (`a[i] == -1`), print "Empty".
   - Otherwise, print the key stored at that index.